Cycle Detection, Order Finding and Discrete Log with Jumps

Sourav Chakraborty[1] David García-Soriano[1] Arie Matsliah[1]
[1]CWI, Amsterdam, Netherlands
sourav.chakraborty@cwi.nl david@cwi.nl A.Matsliah@cwi.nl

Abstract: Let S be a finite set. Given a function $f : S \to S$ and an element $a \in S$, define $f^0(a) = a$ and $f^i(a) = f(f^{i-1}(a))$ for all $i \geq 1$. Let $s \geq 0$ and $r > 0$ be the smallest integers such that $f^s(a) = f^{s+r}(a)$. Determining s and r, given $a \in S$ and a black-box oracle to f, is the *cycle-detection problem*. When f is bijective (i.e., f is a permutation of S), the *order-finding problem* is to find the smallest $r > 0$ such that $f^r(a) = a$, and the *discrete-log problem* is, given an additional element $b \in S$, to find the smallest $k \geq 0$ such that $f^k(a) = b$.
We study the query complexity of these problems with oracles that allow "jumps" to distant positions in the sequence $\bar{a} \triangleq f^0(a)f^1(a)f^2(a)\cdots \in S^*$ at unit cost. Specifically, for every $m \in \mathbb{N}$ the oracle O_f^m is defined, which for every $a \in S$ allows to look ahead at any position $i < m$ in the sequence $\bar{a}$; that is, $O_f^m(a,i) = f^i(a)$ for every $(a,i) \in S \times [m]$.
We show that with an *unrestricted oracle* O_f^∞, the cycle-detection and order-finding problems can be solved using $O(\log s + \log r/\log\log\log r)$ and $O(\log r/\log\log\log r)$ queries, respectively, regardless of $|S|$. This is nearly optimal, as we also prove lower bounds of $\Omega(\log s + \log r/\log\log r)$ and $\Omega(\log r/\log\log r)$ queries. Interestingly, for the discrete-log problem, our results combined with the algorithm of Sutherland [8] imply a lower bound of $\Omega(\sqrt{r}/\log r)$ queries (where r is the size of the cycle to which both a and b belong), which is tight up to the $\log r$ factor. This contrasts with the fact that, with generic group-operation oracles, the problems of order finding and discrete log are known to have polynomially related query complexities.
We also provide algorithms and lower bounds for general oracles O_f^m, $m \in \mathbb{N}$, improving results from earlier work. In particular, with $m = \text{poly}(r)$, our lower bound for order-finding improves the previous bound of $\widetilde{\Omega}(r^{1/3})$ queries, proved by Cleve [2], to $\widetilde{\Omega}(r^{1/2})$, which is nearly optimal.

Keywords: cycle detection, order finding, period finding, query complexity, sublinear algorithms.

1 Introduction

Cycle detection, order finding and discrete log are well-studied problems in various settings and models. There are plenty of algorithms, lower bounds and more general time-space trade-off results known for these problems (some of the highlights can be found on the Wikipedia pages http://en.wikipedia.org/wiki/Cycle_detection and http://en.wikipedia.org/wiki/Discrete_log).

In most of the relevant literature, time and space complexity are the main measures of efficiency for algorithms solving these problems. The classical "tortoise and hare" algorithm of Floyd [3] is probably the best example of a cycle-detecting algorithm with optimal space complexity: it uses only two pointers to elements in S, which move through the sequence $\bar{a} = f^0(a)f^1(a)\cdots$ at different speeds, and detects a cycle after $O(s + r)$ steps (and function evaluations).

In the present paper the main measure of efficiency considered is the query complexity (number of elements of sequence $\bar{a}$ inspected). Clearly, with the standard oracle, which only allows to evaluate f on a certain input, one cannot do better than evaluating f at least $s + r$ times. Here we consider the more powerful oracles, which allow longer "jumps" in the sequence $\bar{a}$ at unit cost.

There are various scenarios in which our objective to minimize the number of such queries may make sense. One example is when S is the set of possible states of a system and f corresponds to a program being executed on it; that is, f maps a given state a to the state $f(a)$ reached on completion of the next execution step. In this setting, running the program for $i > 1$ steps and then reading the state $f^i(a)$ may be almost as fast as reading just the next state $f(a)$.

We are aware of two works that are directly related to the model we study here. First is the decade-old work of Cleve [2], where a query-complexity lower bound is shown for order-finding. Second is the more recent work of Lachish and Newman [5], who study the related problem of periodicity testing.

Also somewhat related are the works in which S corresponds to a group, and the complexity of these problems is measured in terms of the number of group operations required before obtaining the result. See more on this in Section 5.3.

2 Definition of the model and problems

Unless explicitly mentioned otherwise, all indices in this paper are 0-based by default; likewise, $[m] = \{0, 1, \dots, m-1\}$. The symbol log denotes logarithms to the base 2, and ln denotes the natural logarithm. For notational brevity, instead of writing $\max\{\log x, 1\}$, we define $\log x$ to be 1 when $x < 2$ in order for expressions such as $\log\log n$ to be defined for all n.

2.1 The model

Here S is a finite set and f an arbitrary function mapping S to itself. In the unrestricted case we are given an oracle $O_f^\infty : S \times \mathbb{N} \to S$ that maps every query (a, i) to $f^i(a)$. (The iterated function $f^i(a)$ is defined as $f^0(a) = a$ and $f^i(a) = f^{i-1}(f(a))$.) In the m-restricted case, where $m \in \mathbb{N}$, the oracle $O_f^m : S \times [m] \to S$ is defined similarly, except restriction $0 \le i < m$ must hold. When we want to impose the additional constraint that f be a permutation of S, we may write π instead of f.

2.2 The problems

The problems we consider here are:

- **Cycle detection:** Given $a \in S$ and oracle access to f, find the smallest $s \ge 0$ and $r > 0$ such that $f^s(a) = f^{s+r}(a)$. Considering the sequence $\bar{a} = a_0 a_1 \dots$ given by $a_i = f^i(a)$, it is easily seen that $a_0, \dots, a_{r+s-1}$ are distinct and $a_i = a_{i+r}$ whenever $i \ge s$. In this case an equivalent definition avoiding an explicit mention of the function f is an oracle that allows probing a sequence $\bar{a} \in S^*$ having the property that $a_i = a_j$ implies $a_{i+1} = a_{j+1}$. The integer r is called the *length* of the cycle, and s its *starting position.*
- **Order finding:** Given $a \in S$ and oracle access to π, find the smallest $r > 0$ such that $\pi^r(a) = a$; this is the length of the cycle to which a belongs in the cycle decomposition of π. Similarly, one can view this as the problem of finding the period length r in a purely periodic sequence $\bar{a}$, in which $a_0, \dots, a_{r-1}$ are distinct and $a_i = a_{i+r}$ for all $i \ge 0$ (i.e. $s = 0$).[1] The m-restricted oracle is viewed in this setting as allowing one to query position $p + i$ of $\bar{a}$ (where $0 \le i < m$), provided $p = 0$ or is a previously queried position.
- **Discrete log:** Given $a, b \in S$ and oracle access to π, find the smallest $k > 0$ such that $\pi^k(a) = b$. If no such k exists (i.e. a and b belong to different cycles), output ∞.

3 Our results

1) Cycle detection

We show that with the unrestricted oracle O_f^∞, $O(\log s + \log r / \log\log\log r)$ queries are sufficient for cycle detection. Furthermore, if r is promised to be a prime power then $O(\log s + \log r / \log\log r)$ queries suffice. We also show a nearly matching lower bound of $\Omega(\log s + \log r / \log\log r)$ queries for this problem.

For restricted oracles O_f^m we prove an upper bound of $O\left(\log s + s/m + \log r / \log\log\log r + r/\log m\right)$ queries, and a lower bound of

$$\Omega(\log s + s/m + \log r / \log\log r + \\ + \sqrt{r/(\log m \log r)} + r/m)$$

queries.

2) Order finding in permutations

For O_f^∞ we show that $O(\log r / \log\log\log r)$ queries are sufficient for order finding (here too, $O(\log r / \log\log r)$ queries suffice if r is promised to be a prime power), and that $\Omega(\log r / \log\log r)$ queries are necessary.

For the general oracle O_f^m we prove an upper bound of $O\left(\log r / \log\log\log r + r/\log m\right)$ queries, and a lower bound of $\Omega(\log r / \log\log r + \sqrt{r/(\log m \log r)} +$

[1] One may also consider the problem of finding the period of a general sequence (not arising from a permutation), where the same value may appear several times within each period. In this case, upper and lower bounds of $\Theta(r)$ queries are straightforward (for any type of oracle). However, in the property-testing setting, where the task is to distinguish periodic sequences from those that are "far from periodic", highly non-trivial bounds were obtained in [5]

$r/m)$ queries. This improves the earlier bound proved by Cleve [2], which is $\Omega\left(\frac{r^{1/3}}{\sqrt{\log m}}\right)$.

3) Discrete log in permutations

Using a reduction to the generic group computation model of Babai and Beals [1, 8], we obtain as a corollary a lower bound of $\Omega(\sqrt{r}/\log r)$ queries for the discrete-log problem in our model, which is nearly tight. This shows an exponential separation between the query complexities of order finding and discrete log. In the generic group model, however, these two problems have polynomially related query complexities. (In fact, the best separation known is $\Omega(\sqrt{r})$ for discrete log vs $O(\sqrt{r/\log\log r})$ for order finding.)

4 Preliminaries and basic facts from number theory

Our notation is standard, a couple of exceptions aside. We write $\operatorname{lcm}(S)$ for the least common multiple of all elements of a set $S \subseteq \mathbb{N}$ is used. If $a \geq 0, b > 0$, we also write $a \mod b$ for the unique $0 \leq r < b$ such that $a \equiv r \mod b$.

Lemma 4.1(Prime Number Theorem) *Let $\mathcal{P}$ denote the primes and $\mathcal{P}_n \triangleq \mathcal{P} \cap \{1, \ldots, n\}$, where $n \in \mathbb{N}$. Then $|\mathcal{P}_n| = \frac{n}{\ln n} \pm O\left(\frac{n}{(\ln n)^2}\right)$.*

Lemma 4.2(Chinese Remainder Theorem) *Let $m_1, \ldots, m_k, a_1, \ldots, a_k \in \mathbb{Z}$. The system of congruences $x \equiv a_1 \mod m_1, \ldots, x \equiv a_k \mod m_k$ has a unique solution modulo $\operatorname{lcm}(m_1, \ldots, m_k)$ iff for all i, j, $a_i \equiv a_j \mod \gcd(m_i, m_j)$.*

Lemma 4.3 (Divisor bound [4,9]) *Let $\tau(n)$ be the number of positive divisors of $n \in \mathbb{N}$; then $\tau(n) \leq 2^{O(\log n/\log\log n)}$.*

5 Lower bounds for cycle detection and order finding

5.1 Unrestricted oracle

Theorem 5.1

- *Cycle detection with an unrestricted oracle O_f^∞ requires $\Omega(\log s + \log r/\log\log r)$ queries.*
- *Order finding with an unrestricted oracle O_f^∞ requires $\Omega(\log r/\log\log r)$ queries.*

The $\Omega(\log s)$ term in the first item is clear, even under the promise that the period is $r = 1$, since the problem of determining the index of the first '1' in a sequence consisting of s zeroes followed by an infinite number of ones can be reduced to it.[2] So we only need to show a lower bound of $\Omega(\log r/\log\log r)$, which clearly follows from the second item, i.e. the bound for order finding. We sketch here the gist of the proof; a formal proof is given in Appendix A.

Suppose that r is picked at random from $[n/2, n]$. Consider any stage in which the algorithm has queried positions $q_1, \ldots, q_i$ so far, and received answers $\alpha_1, \ldots, \alpha_i$, respectively. Take the set C of *candidate periods* that are consistent with all these query/answer pairs. It is easy to see that this set depends only on the pairs of queries that got the same answer, and not on the labels of the answers themselves. More precisely, C is the set of all $r \in [n/2, n]$ that divide all $q_i - q_j$ for pairs such that $\alpha_i = \alpha_j$ and do not divide any other $q_i - q_j$. Intuitively, what this means is that each of the possible responses to the $(i+1)$-th query falls into at most $i+1$ equivalence classes: the answer can either be one of $\alpha_1, \ldots, \alpha_i$, or be a new one, but other than that the values of $\alpha_1, \ldots, \alpha_i$ themselves are irrelevant. This means that after q queries there are at most $q!$ non-equivalent sets of query/answer pairs received. Hence any algorithm that tries to return the correct value of $r \in [n/2, n]$ and succeeds with constant probability must satisfy $q! \geq \Omega(n)$, from which $q \geq \Omega(\log n/\log\log n) = \Omega(\log r/\log\log r)$ follows.

Remark 5.2 *This is essentially an information-theoretical lower bound, and also holds if r is drawn from any set of size n^ϵ. In particular the lower bound still applies for the special case that r is promised to be a prime power. In this case our algorithms provide a matching $O(\log r/\log\log r)$ upper bound.*

5.2 Restricted oracle

Theorem 5.3 (lower bounds for cycle detection and order finding)

- *Cycle detection with an m-restricted oracle requires*

$$\Omega\left(\log s + \frac{s}{m} + \frac{\log r}{\log\log r} + \sqrt{\frac{r}{\log m \log r}} + \frac{r}{m}\right)$$

[2]Note that it is required here to identify the precise value of s. If an upper bound on s were all that is needed, the query complexity could grow arbitrarily slowly as a function of s (albeit not necessarily of r).

queries.

- *Order finding with an m-restricted oracle requires*

$$\Omega\left(\frac{\log r}{\log\log r}+\sqrt{\frac{r}{\log m \log r}}+\frac{r}{m}\right)$$

queries.

The $\log s$ and $\log r/\log\log r$ terms follow from the analogous bounds for unrestricted oracles. The terms s/m and r/m are clear from the mere fact that the algorithm needs to reach position $s+r$ to detect any cycle. Hence, it suffices to prove a lower bound of $\sqrt{\frac{r}{\log m \log r}}$ queries for order finding. We will prove a seemingly stronger result: setting $n=|S|$, we will show a lower bound of $q(n,m) \triangleq \sqrt{\frac{n}{\log m \log n}}$ queries for order finding under the promise that $r=\Theta(n)$. [3]

For $n \in \mathbb{N}$ and $1<r<n/2$, we denote by $\mathcal{G}_n^r$ the set of all permutations $\pi:[n]\to[n]$ consisting of two disjoint cycles, one of length r and the other of length $n-r$. Given $R \subseteq [n]$, define $\mathcal{G}_n^R \triangleq \bigcup_{r\in R\cap(1,n/2)} \mathcal{G}_n^r$.

Recall that for every permutation π, the m-restricted oracle O_π^m maps $[n]\times[m]$ to $[n]$ according to π; namely, $O_\pi^m(i,j)=\pi^j(i)$. Note that, given access to O_π^m corresponding to some $\pi \in \mathcal{G}_n^{[n]}$, an order finding algorithm should be able to compute r such that $\pi \in \mathcal{G}_n^r$. We prove the lower bound by showing that there is a pair of disjoint sets $R_1, R_2 \subseteq [n/2]$, and a distribution $\mathcal{D}$ on permutations from $\mathcal{G}_n^{R_1} \cup \mathcal{G}_n^{R_2}$, such that no deterministic algorithm can tell if a random $\pi \sim \mathcal{D}$ belongs to $\mathcal{G}_n^{R_1}$ or $\mathcal{G}_n^{R_2}$ unless it makes $\Omega(q(n,m))$ queries to O_π^m.

5.2.1 Formal statement

Let $Q=\{(i_1,j_1),\ldots,(i_q,j_q)\} \subseteq [n]\times[m]$ be a set of q queries (by which we mean here each of the pairs (i,j) fed as input to oracle O_π^m). Let $R_1,R_2 \subseteq (\frac{1}{5}n,\frac{1}{2}n)$ be a pair of disjoint non-empty sets of integers. For $a\in\{1,2\}$, let D_a denote the uniform distribution over all permutations $\pi \in \mathcal{G}_n^{R_a}$.

For the lower bound, it suffices to prove the following:

[3]Cleve [2] proved that the query complexity of order finding with an m-restricted oracle is $\Omega\left(\frac{|S|^{1/3}}{\sqrt{\log m}}\right)$, and if $m \geqslant 2|S|$ then it is $O(|S|^{1/2})$. Since $n=|S|$ is clearly an upper bound on r, we improve Cleve's lower bound by a factor of roughly $\frac{n^{1/6}}{\sqrt{\log n}}$. In particular, for any $m=\mathrm{poly}(n)$ Cleve's bound is $\widetilde{\Omega}(n^{1/3})$ and ours is $\widetilde{\Omega}(n^{1/2})$, which is nearly optimal.

Proposition 5.4 *There are R_1, R_2 with corresponding distributions D_1 and D_2 satisfying the following property: for every (fixed) set $Q=\{(i_1,j_1),\ldots,(i_q,j_q)\}$ of $q=o(q(n,m))$ (distinct) queries and every $\alpha\in[n]^q$, we have*

$$\Pr_{\pi\sim D_1}[O_\pi^m(Q)=\alpha]=(1\pm o(1))\cdot \Pr_{\pi\sim D_2}[O_\pi^m(Q)=\alpha],$$

where $O_\pi^m(Q)$ denotes the string

$$O_\pi^m(i_1,j_1)\cdots O_\pi^m(i_q,j_q)\in[n]^q.$$

5.2.2 Outline of the proof

First, we can assume without loss of generality that any order-finding algorithm finds a collision in π; namely, it makes a pair of queries (i,j) and (i',j') such that $O_\pi^m(i,j)=O_\pi^m(i',j')$. Indeed, once r has been determined, one additional query suffices to find a collision.

Second, we also observe that the actual values returned by oracle O_π^m are irrelevant. Namely, as long as the algorithm finds no collisions, the values obtained from earlier queries are just random elements from $[n]$ (distributed uniformly without repetitions). Therefore, we may assume that the choice of queries is non-adaptive as long as no collision has been found.

Having made these observations, all we need to show is that for any fixed set Q of $o(q(n,m))$ queries and $a\in\{1,2\}$, the probability that Q contains a collision with regard to $\pi\sim D_a$ is $o(1)$.

5.2.3 Core Lemmas

Fix Q as above, and let $Q_1,\ldots,Q_\ell$ be the partition of Q where $(i,j),(i',j')\in Q$ belong to the same Q_h if and only if $i=i'$. Clearly $\ell \leqslant q$. Given π, a subset $Q'\subseteq Q$ is called *π-collision-free* if $O_\pi^m(i,j)\neq O_\pi^m(i',j')$ for all $(i,j)\neq(i',j')\in Q'$. Q' is *r-collision-free* if it is π-collision-free for all $\pi\in\mathcal{G}_n^r$. We say that $Q=Q_1\cup\cdots\cup Q_\ell$ is *component-wise r-collision-free* if Q_h is r-collision-free for every $h\in[\ell]$.

In the following lemmas we let Q be an arbitrary set of size $q=o(q(n,m))$, and by $Q_1,\ldots,Q_\ell$ we denote the foregoing partition of Q.

Lemma 5.5 *For infinitely many $n\in\mathbb{N}$ there exists a pair of non-empty disjoint sets $R_1,R_2\subseteq(\frac{1}{5}n,\frac{1}{2}n)$ such that for $a\in\{1,2\}$,*

$$\Pr_{r\in R_a}\Big[Qiscomponent-wiser{-}collision-free\Big]$$

$$\geqslant 1 - o(1).$$

Given π and $h \neq h' \in [\ell]$, we say that Q_h and $Q_{h'}$ are π-*disjoint* if for all $(i,j) \in Q_h$ and $(i',j') \in Q_{h'}$, $O^m_\pi(i,j) \neq O^m_\pi(i',j')$. We say that Q is π-disjoint if for all $h \neq h' \in [\ell]$, Q_h and $Q_{h'}$ are π-disjoint.

Lemma 5.6 *For every (sufficiently large) n and r, $\frac{1}{5}n < r < \frac{1}{2}n$,*

$$\Pr_{\pi \in \mathcal{G}^r_n}\left[Q is \pi{-}disjoint\right] \geqslant 1 - o(1).$$

Observe that if for some $\pi \in \mathcal{G}^r_n$, Q is both π-disjoint and component-wise r-collision-free, then it is π-collision-free (with regard to that particular π). Hence, by these two lemmas we get the following.

Corollary 5.7 *For infinitely many $n \in \mathbb{N}$ there exists a pair of non-empty disjoint sets $R_1, R_2 \subseteq (\frac{1}{5}n, \frac{1}{2}n)$ such that for $a \in \{1,2\}$,*

$$\Pr_{\pi \in \mathcal{G}^{R_a}_n}\left[Q \textit{ is } \pi{-}collision-free\right] \geqslant 1 - o(1).$$

Proposition 5.4 follows from Corollary 5.7, as sketched in the proof outline.

5.2.4 Proof of Lemma 5.5

We start with an auxiliary lemma.

Lemma 5.8 *There exist absolute constants $\delta > 0$ and $n_0 \in \mathbb{N}$ such that for any $\hat{n} \geqslant n_0$ there is $n = (1 \pm \frac{1}{12})\hat{n}$ and α, β, γ, where $\frac{1}{5} < \alpha < \beta < \gamma < \frac{1}{2}$, for which the following holds. There exist $2k \geqslant \delta n / \log^2 n$ pairs $(p_1,t_1), \ldots, (p_k,t_k), (p'_1,t'_1), \ldots, (p'_k,t'_k)$, such that for all $i \in [k]$ the following holds:*

- *p_i, t_i, p'_i and t'_i are all primes;*
- *$p_i \neq p_j$, $t_i \neq t_j$, $p'_i \neq p'_j$ and $t'_i \neq t'_j$ for all $j \in [k] \setminus \{i\}$;*
- *$p_i + t_i = n$ and $p'_i + t'_i = n$.*
- *$\alpha n < p_i < \beta n$ and $\beta n < p'_i < \gamma n$ (consequently, $p_i < t_i$ and $p'_i < t'_i$);*

Proof. By the Prime Number Theorem, there exists $\epsilon > 0$ and $n_0 \in \mathbb{N}$ such that for any $\hat{n} \geqslant n_0$, the number of primes $\hat{p} \in (\frac{1}{4}\hat{n}, \frac{1}{3}\hat{n})$, as well as the number of primes $\hat{t} \in (\frac{2}{3}\hat{n}, \frac{3}{4}\hat{n})$, is at least $\hat{k} \triangleq \epsilon\hat{n}/\log\hat{n}$. Let $\hat{p}_1, \ldots, \hat{p}_{\hat{k}}$ and $\hat{t}_1, \ldots, \hat{t}_{\hat{k}}$ denote these primes, and consider the multiset $N = \{\hat{p}_i + \hat{t}_j : i,j \in [\hat{k}]\}$. Notice that N contains $\hat{k}^2$ elements, each of them between $\frac{11}{12}\hat{n}$ and $\frac{13}{12}\hat{n}$. Therefore, there must exist some $n \in \mathbb{N}$ appearing in N at least $\ell \triangleq \frac{\hat{k}^2}{\hat{n}/6} = \frac{6\epsilon^2\hat{n}}{\log^2\hat{n}}$ times.

Let $(\hat{p}_{i_1}, \hat{t}_{j_1}), \ldots, (\hat{p}_{i_\ell}, \hat{t}_{j_\ell})$ be the pairs corresponding to this n, namely, $\hat{p}_{i_h} + \hat{t}_{j_h} = n$ for all $h \in [\ell]$. It is clear that $\hat{p}_{i_h} \neq \hat{t}_{j_{h'}}$ for all $h \neq h' \in [\ell]$, since the ranges of $\hat{p}$'s and $\hat{t}$'s are disjoint. Notice that $\hat{p}_{i_h} \neq \hat{p}_{i_{h'}}$ and $\hat{t}_{i_h} \neq \hat{t}_{i_{h'}}$ also hold for all $h \neq h' \in [\ell]$, since all pairs must sum to n. Let β be such that exactly[4] $k \triangleq \ell/2$ of the pairs $(\hat{p}_{i_h}, \hat{t}_{j_h})$ satisfy $\hat{p}_{i_h} < \beta n$.

Denote those k pairs by $(p_1,t_1), \ldots, (p_k,t_k)$, and the remaining k pairs by $(p'_1,t'_1), \ldots, (p'_k,t'_k)$. Let α be such that $\alpha n = \min_{i\in[k]} p_i - 1$, and let γ be such that $\gamma n = \max_{i\in[k]} p'_i + 1$. Clearly, $\alpha < \beta < \gamma$. Since $n \in (\frac{11}{12}\hat{n}, \frac{13}{12}\hat{n})$ and $\max_{i\in[k]} p'_i < \hat{n}/3$ we also have $\gamma < 1/2$. Similarly, $\min_{i\in[k]} p_i \geqslant \hat{n}/4$ and so $\alpha > 1/5$. Setting $\delta = 5\epsilon^2$, the bound $2k \geqslant \delta n/\log^2 n$ follows from $n \in (\frac{11}{12}\hat{n}, \frac{13}{12}\hat{n})$ as well. □

Proof of Lemma 5.5. Let $n \in \mathbb{N}$ be one of those for which Lemma 5.8 holds. Let $R_1 = \{p_1, \ldots, p_k\}$, $T_1 = \{t_1, \ldots, t_k\}$, $R_2 = \{p'_1, \ldots, p'_k\}$ and $T_2 = \{t'_1, \ldots, t'_k\}$. The conditions in Lemma 5.8 imply $R_1, R_2 \subseteq (\frac{1}{5}n, \frac{1}{2}n)$ and $R_1 \cap R_2 = \emptyset$.

Let $a \in \{1,2\}$ and $r \in R_a$. Consider a single component $Q_h = \{(i,j_1), \ldots, (i,j_{|Q_h|})\}$ in the partition of Q. Notice that if Q_h is *not* r-collision-free, then there must be a pair $j \neq j' \in \{j_1, \ldots, j_{|Q_h|}\}$ that satisfies either $j - j' \equiv_r 0$ or $j - j' \equiv_{n-r} 0$ (depending on which cycle contains element i). Let R be the set of all $r \in R_a \cup T_a$ for which some pair j, j' satisfies $j - j' \equiv_r 0$. Since R contains only primes that are greater than $n/5$ and $j \neq j'$, the inequality

$$|j - j'| \geqslant \prod_{r\in R} r \geqslant (n/5)^{|R|}$$

must hold. On the other hand $|j - j'| \leqslant m$, so $|R| \leqslant \frac{\log m}{\log(n/5)}$.

Consequently, the number of different $r \in R_a$ for which *some* pair $j, j' \in \{j_1, \ldots, j_{|Q_h|}\}$ satisfies $j - j' \equiv_r 0$ or $j - j' \equiv_{n-r} 0$ is bounded by $2|Q_h|^2 \frac{\log m}{\log(n/5)}$. This means that for a random $r \in R_a$, the probability that any particular Q_h is not r-collision-free is at most $2\frac{|Q_h|^2 \log m}{|R_a| \log(n/5)}$, and by the union bound,

$$\Pr_{r\in R_a}\left[Q \text{ is } \textit{not} \text{ component-wise } r\text{-collision-free}\right] \leqslant$$

[4]We assume without loss of generality that ℓ is even. If not, drop one pair.

$$\frac{2(\sum_{h\in[\ell]}|Q_h|^2)\log m}{|R_a|\log(n/5)} \leqslant \frac{2|Q|^2\log m}{|R_a|\log(n/5)}.$$

The lemma follows since $|R_a| = \Omega(n/\log^2 n)$ and $|Q| = o\left(\sqrt{\frac{n}{\log m \log n}}\right)$. □

5.2.5 Proof of Lemma 5.6

Let $n \in \mathbb{N}$ be large enough, and let $r \in (\frac{1}{5}n, \frac{1}{2}n)$. Fix a pair of components $Q_h = \{(i, j_1), \ldots, (i, j_{|Q_h|})\}$ and $Q_g = \{(i', j'_1), \ldots, (i', j'_{|Q_g|})\}$ in the aforementioned partition of Q. We now bound the probability, taken over random $\pi \in \mathcal{G}_n^r$, that Q_h and Q_g are *not* π-disjoint.

Notice that when picking a random $\pi \in \mathcal{G}_n^r$, then either i and i' belong to different cycles in π (and therefore Q_h and Q_g are π-disjoint) or otherwise, π locates both i and i' on one of the cycles, where the positions of i and i' are distributed uniformly at random. Both cycles in π are of length greater than $n/5$, hence the probability that Q_h and Q_g are not disjoint is at most $\frac{|Q_h||Q_g|}{n/5}$.

Taking the union bound on all pairs of components we derive an upper bound on $\Pr_{\pi\in\mathcal{G}_n^r}\Big[Q \text{ is not } \pi\text{-disjoint}\Big]$ of

$$\sum_{h\neq g\in[\ell]} \frac{|Q_h||Q_g|}{n/5} \leqslant \frac{(\sum_{h\in[\ell]}|Q_h|)(\sum_{g\in[\ell]}|Q_g|)}{n/5} = 5q^2/n.$$

The lemma follows by plugging in the value of q.

Remark 5.9 *Notice that the lower bound we proved does not work for* any *large enough n. But since the n's for which it works are densely spread (for any $\hat{n} \geqslant n_0$ there is $n = (1 \pm \frac{1}{12})\hat{n}$ for which it works), we can extend the lower bound to work for all sufficiently large n by padding with unit-cycles.*

5.3 Lower bound for discrete log via generic group model

In earlier work the query complexity of group properties has been studied in a different (but related) model – the *generic group* model [1,8]. In this setting, one has access to a "black box" that allows one to find the identity element of the group, compute the inverse of an element, and multiply two elements. The black box returns certain labels to which the algorithm is not allowed to ascribe any meaning, except for equality comparisons (the label determines uniquely the group element, but the precise bijection is a priori unknown to the algorithm).

The best known algorithm to compute the order of an element in this model was found by Sutherland [8]. His algorithm runs in time $O(\sqrt{r/\log\log r})$, where r is the order of the group. In particular, its query complexity is bounded by $O(\sqrt{r/\log\log r})$. A lower bound that is polynomial in r was shown by Babai and Beals [1], and Sutherland shows a lower bound of $\Omega(r^{1/3})$ [8]. In contrast, for the similar problem of discrete log over generic groups there are tight $\Theta(\sqrt{r})$ bounds (the lower bound is by Shoup [7], and the upper bound by Shanks [6]). Therefore, discrete log is strictly harder than order finding in the generic group model, but their complexities are polynomially related. In contrast, there is an exponential separation between the two in our model.

Indeed, given $a \in G$, we know that it is possible to find the order r of the (cyclic) group generated by a with $O(\sqrt{r/\log\log r})$ queries in the generic group model. Afterwards, any of the "jumps" allowed in our model (which are of the form a^i for some $i \geq 0$ and a that was obtained previously) can be simulated with $O(\log r)$ queries to a generic group oracle by the standard logarithmic exponentiation algorithm, after reducing i modulo r. So the existence of an algorithm making q queries for the discrete-log problem in our model implies an algorithm for the discrete-log problem in the generic group model making $O(\sqrt{r/\log\log r} + q\log r)$ queries; by the $\Omega(\sqrt{r})$ lower bound of Shoup, one gets $q = \Omega(\sqrt{r}/\log r)$. This is exponentially larger than the upper bound of $O(\log r/\log\log\log r)$ we prove for the order-finding problem.

6 Upper bounds

Let $\mathcal{PD}(n) \subseteq \mathcal{P}_n$ denote the set of prime divisors of n. Also denote by $\nu_p(x)$ the largest power of p that divides x, and if $D \subseteq \mathcal{P}$, let $\nu_D(x) = \prod_{p\in D}\nu_p(x)$, which is the part of the prime factorization of x that uses primes in D. We will make repeated use of the following bound on the size of $\mathcal{PD}(n)$:

Lemma 6.1 $|\mathcal{PD}(n)| \leq 2\log n/\log\log n$.

Proof. This essentially follows from Lemma 4.3, but can also be shown directly: let b be the least integer satisfying $b^b \geq n$; then $b \leq 1 + 2\log n/\log\log n$. There are at most $b-2$ primes in $[2, b-1]$, and since

$b^b \geq n$ there are at most b primes dividing n in $[b, n]$. Hence at most $2b - 2$ primes divide n. □

Throughout this section we assume that the starting position s of the cycle is known (or equivalently $s = 0$), and also that we have an upper bound of n on the cycle length r. In Appendix D we show how to get rid of these assumptions.

Theorem 6.2 *Assume we are given upper bounds n and s on the cycle length r and the starting position s, respectively. Then*

- *there is a randomized algorithm that finds r by making $O(\log n/\log\log\log n)$ queries to the unrestricted oracle O_f^∞;*
- *for any $m \geqslant 1$, there is a deterministic algorithm that finds r with $O(n/m + \sqrt{n})$ queries to the m-restricted oracle O_f^m;*
- *for some fixed $c > 0$, if $m \geqslant 2^{c\sqrt{n}}$, then there is a randomized algorithm that finds r with $O(\frac{n}{\log m} + \frac{\log n}{\log\log\log n})$ queries to the m-restricted oracle O_f^m.*

Observe that for $m \leq 2^{O(\sqrt{n})}$, $\sqrt{n} = O(n/\log m)$, so the bound of the third item still holds, but is weaker than that of the second item in this case. We proceed to present these algorithms.

6.1 Unrestricted oracle

To start with, observe that it is simple to determine, given a prime power p^α, whether p^α divides r using only 1 query. This is because r divides t iff $query(s) = query(t + s)$. So if t is the largest divisor of $\mathrm{lcm}([n])$ that is not a multiple of p^α, then $query(s) \neq query(t + s)$ implies that r does not divide t and this means (by our choice of t) that r must be divisible by p^α.

Algorithm 1 (is_a_divisor(p^α, s, n) – outputs **true** if r is divisible by p^α, where p is prime)

```
1: find the prime factorization of $\mathrm{lcm}([n]) = p_0^{\beta_0} p_1^{\beta_1} \ldots p_t^{\beta_t}$ (where $p_0 = p$)
2: $t \leftarrow p^{\alpha-1} \prod_{i \geqslant 1} p_i^{\beta_i}$
3: return true iff $query(s) \neq query(t + s)$
```

If p is a prime factor of r, using binary search and Algorithm is_a_divisor we can find the exact exponent α of p in the prime factorization of r by making at most $1 + 2\log(\alpha)$ queries; Algorithm find_exponents(D, s, n) in Appendix B.2 does this explicitly for all prime factors of r (assuming $D = \mathcal{PD}(r)$) and returns r itself.

Thus, if we knew somehow the set $\mathcal{PD}(r)$, we could find the precise value of r with at most $\sum_i^k (1 + 2\log(\alpha_i))$ additional queries, where $r = \Pi_i^k p_i^{\alpha_i}$ has been written according to its prime factorization. Claim B.1 in Appendix B.1 shows that this quantity is bounded by $O(\log r/\log\log r)$. Thus the main task is finding the set $\mathcal{PD}(r)$ using $O(\log n/\log\log\log n)$ queries. We divide this task into two. First we present an algorithm (find_all_divisors) that, given a set D of primes, makes $O(1 + d\log|D|/\log\log|D|)$ queries and with high probability outputs $\mathcal{PD}(r) \cap D$, where $d = |\mathcal{PD}(r) \cap D|$. In the second part we partition $[n]$ into intervals of increasing length and find the set of prime factors of r in each of them. These intervals will be carefully chosen so as to guarantee that, if we use Algorithm find_all_divisors for each, the overall query complexity of finding all the prime factors of r remains $O(\log n/\log\log\log n)$.

For the first part, the idea is to split the set D into two sets D_1 and D_2 such that if D has at least two prime divisors of r, then both D_1 and D_2 contain at least one prime divisor of r. To this end, we randomly partition D into D_1 and D_2 and check if both D_1 and D_2 have a non-empty intersection with $\mathcal{PD}(r)$. If not, we repeat the process. If $|D \cap \mathcal{PD}(r)| \geqslant 2$, then the probability that a random partition has at least one prime on each side is at least $1/2$. This means that if $|D \cap \mathcal{PD}(r)| \geqslant 2$ then with high probability we partition D into D_1 and D_2 such that both $|D_1 \cap \mathcal{PD}(r) \geqslant 1|$ and $|D_2 \cap \mathcal{PD}(r)| \geqslant 1$ hold. Then we proceed to find the primes in each part recursively.

In order to be able to implement this idea we need two procedures: (a) given a set D, determine whether $D \cap \mathcal{PD}(r) > 0$ using one query, and (b) given D such that $|D \cap \mathcal{PD}(r)| = 1$ find $D \cap \mathcal{PD}(r)$ using only $\log|D|/\log\log|D|$ queries.

The first one is a straightforward generalization of Algorithm is_a_divisor. Thus we move the pseudocode for Algorithm has_a_factor(D, s, n) to Appendix B.3.

Now we present the second procedure. If we are given a subset D that is known to contain *exactly* one prime divisor of r, it is not difficult to see that binary search and Algorithm has_a_factor can be used to find p with $O(\log|D|)$ queries. Unfortunately, this is too expensive for our purposes; we show that we can do with only $O(\log|D|/\log\log|D|)$ queries:

Lemma 6.3 *Algorithm* find_unique_prime_divisor

Algorithm 2 (find_unique_prime_divisor(D, s, n)– finds the unique prime factor of r in D)

```
Require: $r \le n$ and $|\mathcal{PD}(r) \cap D| = 1$
1. $Rest \leftarrow \mathcal{P}_n \setminus D$
2. $a \leftarrow \nu_{Rest}(\mathrm{lcm}([n]))$
3. $k \leftarrow 1 + 2\lfloor \log|D| / \log\log|D| \rfloor$
4. for $i = 0$ to $k - 1$do
5.    $v_i \leftarrow query(a \cdot i + s)$
6.    if $i > 0$ and $v_i = v_0$ then
7.       return the only prime divisor of $i$
8.    end if
9. end for
10.
11. while $|D| \ge 2$ do
12.    split $D$ into $k$ disjoint sets $D_1, \ldots, D_k$ of
         equal size (up to $\pm 1$)
13.    let $a_i \leftarrow a \cdot \nu_{D_i}(\mathrm{lcm}([n]))$ for all $i \in [1, k]$
14.    find $x$ such that $x \equiv i \cdot a \mod a_i$ for all $i \in [1, k]$
15.    $y \leftarrow query(x + s)$
16.    find $1 \le i \le k$ such that $v_i = y$
17.    $D \leftarrow D_i$
18. end while
19. return the unique $p \in D$
```

(D,s,n) *finds the only* $p \in \mathcal{PD}(r) \cap D$ *making* $O(\log|D|/\log\log|D|)$ *queries.*

Proof. First we show that the query complexity is $O(\log|D|/\log\log|D|)$. Clearly the first **for** loop makes $O(\log|D|/\log\log|D|)$ queries. At the start of the **while** loop, $|D| \le k^k$, and every iteration divides the size of D by a factor of k. So the loop runs for at most k iterations and since only one query is made inside each iteration the while loop makes $O(\log|D|/\log\log|D|)$ queries in total.

Now we show the correctness of the algorithm. Let $\mathcal{PD}(r) \cap D = \{p\}$ and the exponent of p in r be t. Clearly if p^t is less than k, the first **for** loop will find it when i reaches the value p^t, and not before. So all we need to show is that if $p^t \geqslant k$, the **while** loop finds the correct D_i to which p belongs.

First of all, the **while** loop needs to find an x such that $x \equiv i \cdot a \mod a_i$ for all i. The existence of such an x is guaranteed by the Chinese Remainder Theorem (some of the sets D_i may be empty, in which case $a_i = a$). If $p \in D_i$, then p^t divides $\nu_{D_i}(\mathrm{lcm}([n]))$ and hence r divides a_i. So $x \equiv i \cdot a \mod r$ and this implies $query(x + s) = query(i \cdot a + s) = v_i$. To complete the proof we have to show that no other $j \neq i$ can satisfy $query(x+s) = query(j \cdot a + s)$, or equivalently $x \equiv j \cdot a \mod r$. This is because $x \equiv j \cdot a \mod r$ and $x \equiv i \cdot a \mod r$ together imply $a(i - j) \equiv 0 \mod r$, and since $p^t \mid r$ and $\gcd(a, p^t) = 1$ we get $i \equiv j \mod p^t$ and $k > |j - i| \ge p^t$, which is a contradiction. Hence the **while** loop does indeed find the unique i such that $p \in D_i$. □

We will also need an algorithm has_two_prime_divisors that determines with high probability whether a given set D contains at least two prime factors of r. Such algorithm can be designed along the same lines as Algorithm find_unique_prime_divisor. The details are worked out in Appendix B.4, Lemma B.2.

Now that we have both of the necessary procedures we present the recursive algorithm for finding $D \cap \mathcal{PD}(r)$.

Algorithm 3 find_all_divisors(D, s, n) – finds all *prime* divisors of r in D

```
Require: $r \le n$ and $D \subseteq \mathcal{P}_n$ is a set of primes
1: if has_two_prime_divisors(D, s, n) then
2:    repeat
3:       split $D$ into two sets $D_1$ and $D_2$ at random
         (i.e. each $x \in D$ lands into $D_1$ with probability
         1/2)
4:    until has_a_factor($D_1$, s, n) and has_a_factor
         ($D_2$, s, n)
5:    return find_all_divisors($D_1$, s, n)$\cup$find_all_divisors
            ($D_2$, s, n)
6: else
7:    if has_a_factor(D, s, n)then
8:       return {find_unique_prime_divisor(D, s, n)}
9:    else
10:      return $\emptyset$
11:   end if
12: end if
```

Lemma 6.4 *If* $D \subseteq \mathcal{P}_n$ *contains* d *prime factors of* r *and* $|D| \geqslant \log n$, *then with probability at least* $(1 - 2d\log\log n/(1000\log n))$, *Algorithm* find_all_divisors(D, s, n) *manages to find* $\mathcal{PD}(r) \cap D$. *Its expected query complexity is* $O(d\log|D|/\log\log|D|)$.

(The procedure also trivially works with $O(1)$ queries if D is empty).

Proof. Supposing all calls to Algorithm has_two_prime_divisors return the correct answer, it is easy to see that find_all_divisors(D, s, n) terminates and outputs the set $D \cap \mathcal{PD}(r)$. Thus all we need to check is the query complexity and the probability of has_two_prime_divisors making no errors.

Since each call to find_all_divisors(D, s, n) either splits D into two disjoint sets each containing at least one prime factor of r or returns after a call to find_unique_prime_divisor, the total number of (recursive) calls to find_all_divisors is at most $2d - 1$ (in fact it is exactly $2d - 1$ if no mistake is made). So the total number of queries spent on calls to has_two_prime_divisors, has_a_factor and find_unique_prime_divisor is at most $O((2d-1)(1 + \log\log n/\log\log\log n) + d + d\log|D|/\log\log|D|) = O(d\log|D|/\log\log|D|)$ since $|D| \geq \log n$. Also recall that has_two_prime_divisors never returns **true** unless $|D \cap \mathcal{PD}(r)| \geq 2$. For any such D, the expected number of tries before a successful split in the **repeat** loop is at most 2, hence the total expected query complexity of all executions of this loop is bounded by $2(d-1)$. This shows that the overall expected query complexity is $O(d\log|D|/\log\log|D|)$ if $|D| \geq \log n$.

We turn now to bounding the error probability. Each call to has_two_prime_divisors is incorrect with probability at most $1/(1000\log n)$ by Lemma B.2. Hence by the union bound the error probability is at most $(2d-1)/(1000\log n)$. □

Algorithm find_all_divisors still falls short of our upper bound of $O(\log n/\log\log\log n)$ queries, since an invocation with $D = [n]$ might require $\Omega((\log n/\log\log n)^2)$ queries on average (as r can have as many as $\Omega(\log n/\log\log n)$ prime divisors). To overcome this difficulty, we partition $[n]$ into consecutive intervals of increasing length; the intuition is being that if $a < b < c < d$, the maximum number of prime divisors of r interval $[c, d]$ can contain is smaller than the number of divisors of r interval $[a, b]$ can contain. The key is to choose the right division of $[n]$ into a sequence of intervals.

Lemma 6.5 *Algorithm* find_period(s, n) *makes* $O(\log n/\log\log\log n)$ *expected queries and with probability at least* $1 - 1/250$ *returns* r.

Note that standard techniques can be used to turn this algorithm into one with a worst-case $O(\log n/\log\log\log n)$ query complexity guarantee and, say, $1-1/100$ success probability. Namely, it suffices to stop the call find_period(s, n) after the number of queries made is 2.5 times larger than its expected query complexity.

Proof. First we show correctness. Note that $a < b = \Theta(\log n/\log\log n)$. Define the sequence $l_i \triangleq 2^{i\log i}$; it is easy to see that $l_a = \Theta(\log n)$ and $l_{b-1} = \Theta(n)$. The interval $[n]$ is partitioned into $O(\log n/\log\log n)$ parts: namely $[0, l_a - 1]$, $[l_b, n]$ and the intervals $[l_i, l_{i+1} - 1]$ for $a \leq i < b - 1$. All prime factors in the range $[l_{b-1}, n]$ are found in the third line; all prime factors in the interval $[2, l_a - 1]$ are found in the first **for** loop; all prime factors in $[l_a, l_{b-1})$ are contained in one of the intervals $[l_i, l_{i+1} - 1]$ taken care of in the second **for** loop.

For each interval, Algorithm find_all_divisors is called at most once and from Lemma 6.4 and the union bound we obtain that with probability at least $(1 - 2d\log n\log n/(1000\log n))$ all the answers of all the calls to find_all_divisors are correct, where $d \triangleq |\mathcal{PD}(r)|$ is the number of prime divisors of r. We know that $d \leq 2\log n/\log\log n$, so with probability $1-1/250$ the algorithm gains knowledge of the set of all the prime divisors of r on completion of the loop, and then the call to find_exponents(PD, s, n) returns the period.

Now we prove the bound on the query complexity. The fact that $l_b = \Theta(n)$ implies that at most $O(1)$ prime divisors of $r \leq n$ can be in the interval $[l_b, n]$, so the expected number of queries made for this interval is $O(\log n/\log\log n)$. Also $l_a = \Theta(\log n)$, hence by the prime number theorem there are $\Theta(\log n/\log\log n)$ primes in the interval $[0, l_a - 1]$. Thus the first loop of Algorithm is_a_divisor makes a total of $O(\log n/\log\log n)$ queries.

As for the second **for** loop, let k_i be the size of the interval $[l_i, l_{i+1} - 1]$ and n_i be the number of prime factors of r in the interval $[l_i, l_{i+1} - 1]$; that is, $k_i \triangleq l_{i+1} - l_i = \Theta(l_{i+1})$ and let $n_i \triangleq |\mathcal{PD}(r) \cap [l_i, l_{i+1} - 1]|$. By Lemma 6.4 and linearity of expectation, it follows that the expected number of queries made by Algorithm find_all_divisors in the second loop is $O(\sum_{i=1}^{b-1} n_i \log k_i/\log\log k_i)$. Note that

$$n \geqslant r \geqslant \prod_{i=1}^{b-1} l_i^{n_i} = \prod_{i=1}^{b-1} 2^{n_i\, i\log i},$$

so by taking logarithms on both sides we obtain

$$\sum_{i=1}^{b-1} n_i\, i\log i \leq \log n.$$

Clearly for all $i \geqslant a$ we have $k_i \geqslant \Theta(\log n)$. These inequalities can be used to bound the total expected number of queries made in the second **for** loop by

$$\sum_{i=a}^{b-2} n_i \frac{\log k_i}{\log\log k_i} \leq \frac{O(1)}{\log\log\log n} \sum_{i=a}^{b-2} n_i\, i\log i \leq O\left(\frac{\log n}{\log\log\log n}\right),$$

as we wished to show. □

Algorithm 4 find_period(s, n)–finds r

```
Require: the period has length $\le n$ and starts at position $\le s$
1: $a = \min\{i \mid 2^{i \log i} \ge \log n\}$
2: $b = \min\{i \mid 2^{i \log i} > n\}$
3: $\mathcal{PD} = \text{find\_all\_divisors}([2^{(b-1)\log(b-1)}, n], s, n)$
4: for all $p \in \mathcal{P}_{2^{a \log a}-1}$ do
5:     if is_a_divisor$(\{p\}, s, n)$ then $\mathcal{PD} \leftarrow \mathcal{PD} \cup \{p\}$
6:
7:     end if
8: end for
9:
10: for $i = a$ to $b - 2$ do
11:     $I = [2^{i \log i}, 2^{(i+1)\log(i+1)} - 1]$
12:     $\mathcal{PD} \leftarrow \mathcal{PD} \cup \text{find\_all\_divisors}(I, s, n)$
13: end for
14: return find_exponents$(\mathcal{PD}, s, n)$
```

6.2 Restricted oracle

To deal with the restricted-oracle case, we first make the following simple observation, the proof of which is in Appendix C.1.

Observation 6.6 *Let $\mathcal{A}$ be an adaptive algorithm that finds r by making q queries to the unrestricted oracle and let g be the largest position that $\mathcal{A}$ queries. Then for any $m > 0$ there is an algorithm that finds r with $q + \frac{g}{m}$ queries to the m-restricted oracle.*

We present two algorithms for the restricted-m case. While the first one is better when $m < 2^{O(\sqrt{n})}$, the second one gives better query complexity for $m \ge 2^{\Omega(\sqrt{n})}$.

6.2.1 Restricted oracle, for any $m > 0$

This algorithm uses the baby steps-giant steps method of Shanks [6,8].

Lemma 6.7 *Let $n > 0$ and define $a \triangleq \lceil \sqrt{n} \rceil$, $B \triangleq \{0, 1, 2, \ldots, a-1\}$ and $G \triangleq \{a, 2a, 3a, \ldots, a^2\}$. If $r \le n$, then there are $b \in B$ and $g \in G$ such that $g - b = r$.*

Proof. Clear if $r \in G$. Otherwise write $r = aq + t$, where $0 \le q < a$ and $0 < t < a$. Taking $b = a - t \in B$ and $g = a(q+1) \in G$, we have $r + b = g$. □

Algorithm find_r_small_m(s, n) needs to make $|B| + |G| = O(\sqrt{n})$ queries. Note that the maximum position queried is $a^2 \le n + 2\sqrt{n}$. So from Observation 6.6 we have the following lemma.

Lemma 6.8 *For any $m > 0$ there is a deterministic, non-adaptive algorithm that makes $O(\sqrt{n} + \frac{n+s}{m})$ queries to an m-restricted oracle and outputs r.*

Algorithm 5 find_r_small_m(s, n)-find r

```
1: $a \leftarrow \lceil \sqrt{n} \rceil$
2: for $i \in \{0, \ldots, a-1\}$ do
3:     $b_i \leftarrow i + s$
4:     $g_i \leftarrow (i+1) \cdot a + s$.
5: end for
6: query $\cup_i \{b_i, g_i\}$
7: return the smallest positive value of $(a \cdot (j+1) - i)$ among all $i, j$ such that
   $query(b_i) = query(g_j)$.
```

6.2.2 Restricted oracle for $m \ge 2^{\Omega(\sqrt{n})}$

To begin we need the following lemma (the proof is in Appendix C.2):

Lemma 6.9 *Let c be a large enough constant. If $m \ge 2^{c\sqrt{n}}$, there exists a partition of $\mathcal{P}_n$ into $A_0, \ldots, A_k$ with the following properties:*

1. *$A_0 \triangleq \mathcal{P}_n \cap [1, \sqrt{n}]$;*
2. *for all $0 \le i \le k$, $\nu_{A_i}(\text{lcm}([n])) < \sqrt{m}$.*
3. *$k = O(n / \log m)$.*

Algorithm 6 find_r_large_m(s, n)–find r when $m \geqslant 2^{c\sqrt{n}}$

```
1: find the partition of $\mathcal{P}_n$ into $A_0, \ldots, A_k$
   (as in Lemma 6.9)
2: for $i = 1$ to $k$ do
3.     if $query(\nu_{A_0 \cup A_i}(\text{lcm}([n])) + s) = query(s)$
       then
4:         $N \leftarrow \nu_{A_0 \cup A_i}(\text{lcm}([n]))$
5:         use the algorithm find_period(s, n) from Section
            6.1 to find $r$, replacing each query to position $i$
           with a query toposition $(i \mod N)$.
6:         return $r$
7:     end if
8: end for
9:
```

Lemma 6.10 *Let c be as in Lemma 6.9. If $m \geqslant 2^{c\sqrt{n}}$ then there Algorithm 6 makes $O(s/m + n/\log m + \log n / \log\log\log n)$ queries to an m-restricted oracle and outputs the period r.*

Proof. The first property of the partition implies that at most one A_i with $i > 0$ contains a prime divisor of r. In any case the **for** loop will find an i such that $r \mid \nu_{A_0 \cup A_i}(\text{lcm}([n]))$. So after a suitable i has been found, we know that r divides $N \triangleq \nu_{A_0 \cup A_I}(\text{lcm}([n])) < \sqrt{m} \times \sqrt{m} = m$ by the second property, and therefore for all i, if $i \equiv i'(mod\ N)$ then $query(i) = query(i')$. Hence by using the algorithm of Section 6.1, with each query to position i replaced by a query to position $(i \mod N)$, r is found with $O(\log n/\log\log\log n)$ additional queries.

It is clear that the main loop spends at most $O(k) = O(n/\log m)$ queries (by the third property) on checking the **if** condition inside the **for** loop. So the total number of queries made by find_r_large_m(s, n) is $O(n/\log m + \log n/\log\log\log n)$. Also note that the maximum position the algorithm queries is $s + \max_i\{\nu_{A_0\cup A_i}(\text{lcm}([n]))\} < s + m$. So from Observation 6.6 we have an algorithm that finds r by making at most $O(s/m+n/\log m+\log n/\log\log\log n)$ queries to an m-restricted oracle. □

Acknowledgement

We are grateful to Ronald de Wolf for introducing us to the problem of order finding with jumps, and for many valuable discussions and comments. We also thank Richard Cleve for e-mail correspondence.

References

[1] L.Babai and R. Beals. A polynomial-time theory of black-box groups. In *Groups St Andrews 1997 in Bath, I, pp. 30-64. London Math. Soc. Lect. Notes* 260, 1999.

[2] R. Cleve. The query complexity of order-finding. In *Proceedings of 15th IEEE Conference on Computational Complexity*, pages 54-59, 2000.

[3] R.W. Floyd. Nondeterministic algorithms. *J. ACM,* 14(4):636-644, 1967.

[4] G. H. Hardy and E. M. Wright.*An Introduction to the Theory of Numbers.* Oxford University Press, New York, fifth edition, 1979.

[5] O. Lachish and I. Newman. Testing periodicity. *Algorithmica*, 2009. Earlier version in RANDOM'05.

[6] D. Shanks.Class number, a theory of factorization, and genera. In *Analytic Number Theory, Proceedings of Symposia on Pure Mathematics,* volume 20, pages 415-440, 1971.

[7] V.Shoup. Lower bounds for discrete logarithms and related problems. In *EUROCRYPT'97: Proceedings of the 16th annual international conference on Theory and application of cryptographic techniques*, pages 256-266, Berlin, Heidelberg, 1997. Springer-Verlag.

[8] A. V. Sutherland.Order computations in generic groups. Technical report, PhD Thesis MIT, Submitted June 2007, 2007.

[9] S. Wigert.Sur l'order de grandeur du nombre des diviseurs d'un entier. *Arkiv för Matematik, Astronomi och Fysik*, 3:1-9, 1907.

Appendix A

A Lower bound for order finding with unrestricted oracle

Consider the order-finding problem for a sequence [5] $a = a_f \in [n]^n$ for some $f : [n] \to [n]$ We say that two subsequences $a, b \in [n]^{\leq n}$ are *equivalent* if they are the same up to a relabelling, i.e. if there is a permutation $\sigma : [n] \to [n]$ such that $\{b_i\}_{i\in[n]} = \{\sigma(a_i)\}_{i\in[n]}$. For each equivalence class $[a]$, choose one representative, which we denote by $[a]$ as well. Then a and b are equivalent iff $[a] = [b]$. For any permutation $\pi : [n] \to [n]$, we can consider also the representative function $[]_\pi$ mapping a to $[a]_\pi \triangleq \pi[a]$.

Take any randomized algorithm $\mathcal{A}$ for order finding that always makes q queries (some of which may be redundant). At any given stage, its behaviour is determined by its random seed t and the history sequence $H = ((q_0, \alpha_0), \ldots, (q_{i-1}, \alpha_{i-1}))$ of query/answer pairs received so far (where $\alpha_i = a_{q_i}$). H starts out empty. On history H, where $i = |H| < q$, $\mathcal{A}$ queries position $q_i \triangleq \mathsf{query}_{\mathcal{A},t}(H)$, obtains $\alpha_i \triangleq a_{q_i}$ as response, and then the new pair (q_i, α_i) is appended to H. After the last query, $\mathcal{A}$ makes a guess for the period of a. For any a, the value returned by $\mathcal{A}$ is correct with probability at least 2/3. In particular, this also holds when $\mathcal{A}$ runs on the "normalized" sequence $[a]$.

We can define a new algorithm $\mathcal{B}$ that simulates $\mathcal{A}$, except that it normalizes on the fly the elements of a it sees according to a random representative function $[]_\pi$, so that the sequence $\alpha_0, \ldots, \alpha_{i-1}$ of responses kept in H is always normalized. Concretely, $\mathcal{B}$ takes the random seed t and an additional uniformly random permutation π, and on history H does the following:

1. Make the same query q_i that $\mathcal{A}$ would make on history H and seed t; let $\alpha = a_{q_i}$ be its answer.
2. Find the unique β satisfying

$$[(\alpha_0, \ldots, \alpha_{i-1}, \alpha)]_\pi = (\alpha_0, \ldots, \alpha_{i-1}, \beta).$$

[5] Strictly speaking, a_f was defined before as an infinite sequence in $[n]^*$, but it is enough to restrict attention to finite sequences of length n

3. Append (q, β) to H (that is, pretend that the answer received was β instead of α).

It is not difficult to see that $\mathcal{B}_{t,\pi}$ also has success probability 2/3 for any a (over random t and π), as its success probability for a is an average of that of $\mathcal{A}$ permutations of a (all of which have the same period). It also has query complexity q and, by construction, its decisions depend only on t and $[a]_\pi$, and therefore once the randomness has been fixed it behaves the same for equivalence sequences; we say that $\mathcal{B}$ is in "normal form".

Proposition A.1 *Order finding requires at least $\Omega(\log r/\log\log r)$ queries.*

Proof. We prove that for every n, order finding under the promise that the period r belongs to some known set $S_n \subseteq [n/2, n]$ of polynomial density (i.e. $l \triangleq |S_n| = n^{\Omega(1)}$) requires $\Omega(\log l/\log\log l) = \Omega(\log r/\log\log r)$ queries.

It is enough to consider algorithms in standard form. Consider the distribution over sequences $a \in [n]^n$ defined by picking a period r uniformly at random from S_n and defining $a_i = (i \mod r)$. As customary, we apply Yao's principle [6]. We show that no deterministic decision tree of depth q in standard form can succeed with probability at least 2/3 over this distribution unless $B_q \geq (2/3)\, l$, where Bell's number B_q is the number of partitions of a set with n elements. It is not difficult to see that $B_q \leq q!$, so this will imply $q = \Omega(\log l/\log\log l)$, as desired.

Now consider a decision tree $\mathcal{T}$ in standard form and let l' be its number of leaves. To leaf number $i \in [l']$ corresponds the set C_i of periods r such that, on the (unique) sequence of period r in the distribution, leaf i is reached. Namely, if the history leading to leaf i is $H = ((q_0, \alpha_0), \ldots, (q_{q-1}, \alpha_{q-1}))$, then $C_i = \{r \in S_n : \forall a, b \in [q], (r \mid (q_a - q_b)) \leftrightarrow \alpha_a = \alpha_b\}$. Note that $l' \leqslant B_q$, because C_i is determined by a partition of $[q]$, where the indices of queries that received the same answer are put into the same set of the partition.

The family $\{C_i\}_{i\in[l']}$ forms a partition of a subset of S_n. Since the tree is deterministic, there is only one $r = r_i$ in each C_i for which the correct period r_i is returned. As r is chosen at random from a set of size l, the probability of picking some $r \in \bigcup_{i\in[l']}\{r_i\}$ is exactly l'/l. Hence for the success probability to be no smaller than 2/3, we need to have $l' \geq (2/3)l$. □

[6] Note that there is a deterministic algorithm that makes *one* query and always works for this particular distribution: first query $n! - 1$ to get $\alpha = (n! - 1) \mod r$, and then return $\alpha + 1$. However, this is *not* in standard form. The point is that any algorithm that works for general sequences can be put into standard form, and to prove lower bounds for those it is enough to restrict ourselves to sequences of this kind.

B Upper bound for unrestricted oracle

B.1 Claim B.1

Claim B.1 *If $\Pi_{i=1}^k p_i^{\alpha_i}$ is the prime factorization of r, then $\sum_{i=1}^k (1 + 2\log(\alpha_i)) = O(\log r/\log\log r)$.*

Proof. The number of divisors of r is $\tau(r) = \prod_i(\alpha_i + 1)$. By Lemma 4.3, $\tau(r) = 2^{O(\log r/\log\log r)}$, from which we get $\sum_i \log\alpha_i = \log\prod_i \alpha_i < \log\tau(n) = O(\log r/\log\log r)$. Since $k = |\mathcal{PD}(r)| = O(\log r/\log\log r)$, we get

$$\sum_{i=1}^k (1 + 2\log\alpha_i) = k + 2\sum_i^k \log\alpha_i = O(\log r/\log\log r).$$

□

B.2 Algorithm find_exponents

Algorithm 7 (find_exponents(D, s, n) – returns r, given the set D of its prime divisors)

```
Require: $D = \mathcal{PD}(r)$
1: $r \leftarrow 1$
2: for all $p \in D$ do
3:    $\alpha, \beta \leftarrow 1$
4:    while is_a_divisor($\{p^\beta\}$, s, n) do
5:       $\beta = 2 \cdot \beta$
6:    end while
7:
8:    while $\beta - \alpha \neq 1$ do
9:       $\gamma \leftarrow \lfloor \frac{\alpha+\beta}{2} \rfloor$
10:      if is_a_divisor($\{p^\gamma\}$, s, n) then
11:         $\alpha = \gamma$
12:      else
13:         $\beta = \gamma$
14:      end if
15:   end while
16:   $r \leftarrow r \cdot p^\alpha$
17: end for
18: return $r$
```

B.3 Algorithm has_a_factor

Algorithm 8 (has_a_factor(D, s, n) – outputs **true** if r is divisible by some element of D)

```
Require: $D = \{p_1^{\alpha_1}, \dots, p_k^{\alpha_k}\}$ ($\alpha_i \ge 1$) is a set of powers of distinct primes; $r \le n$
1: find the prime factorization of $\mathrm{lcm}([n]) = p_1^{\beta_1} \dots p_m^{\beta_m}$ (we order the factors so
   that the first $k$ correspond to the primes in $D$)
2: $t \leftarrow \prod_{i=1}^{k} p_i^{\alpha_i - 1} \prod_{i=k+1}^{m} p_i^{\beta_i}$
3: return true if $query(s) \neq query(t+s)$
```

B.4 Algorithm has_two_prime_divisors

Algorithm 9 has_two_prime_divisors(D, s, n) – determine if D contains ≥ 2 prime divisors of r

```
Require: $r \le n$
1: $Rest \leftarrow \mathcal{P}_n \setminus D$
2: $a \leftarrow \nu_{Rest}(\mathrm{lcm}([n]))$
3: $k \leftarrow 10 + 2\lceil \log\log n / \log\log\log n \rceil$
4:
5: $factors \leftarrow \emptyset$
6: for $i = 0$ to $k-1$ do
7:    $v_i \leftarrow query(a \cdot i + s)$
8:    if $i$ is prime and is_a_divisor$(i, s, n)$ then
9:       $factors = factors \cup \{i\}$
10:   end if
11: end for
12: if $factors \neq \emptyset$ then
13:    return $|factors| \ge 2$ or $|factors| = 1$ and
             has_a_factor(D \ factors, s, n)
14: end if
15:
16: for $i = 1$ to $k$ do
17:    split $D$ into $k$ disjoint sets $D_1, \dots, D_k$ by
        placing each $p \in D$ in a randomly selected $D_i$
18:    let $a_i \leftarrow a \cdot \nu_{D_i}(\mathrm{lcm}([n]))$ for all $i \in [1, k]$
19:    find $x$ such that $x \equiv i \cdot a \mod a_i$ for all
        $i \in [1, k]$
20:    $y \leftarrow query(x + s)$
21:    if there is no $1 \le i \le k$ with $v_i = y$ then
22:       return true
23:    end if
24: end for
25: return false
```

Lemma B.2 *If D contains fewer than 2 prime factors of r, Algorithm* has_two_prime_divisors(D, s, n) *always returns* ***false****. Otherwise it returns* ***true*** *with probability at least $1 - 1/(1000 \log n)$. Its query complexity is $O(\log\log n / \log\log\log n)$.*

Proof. Clearly, if $|D \cap \mathcal{PD}(r)| \le 1$ then the algorithm always returns **false**. Also, the correct decision is always made if there is some element of $\mathcal{PD}(r)$ smaller than $k = 10 + \log\log n / \log\log\log n$.

So assume that $|D \cap \mathcal{PD}(r)| \ge 2$ and $\mathcal{PD}(r) \cap [k] = \emptyset$. Take any pair of distinct $p, q \in D \cap \mathcal{PD}(r)$; with probability at least $1 - 1/k$, they fall into different sets D_i, D_j, where $1 \le i, j \le k$. We claim that whenever this happens, $x \not\equiv (m \cdot a) \mod r$ for any $1 \le m \le k$. Indeed, suppose for a contradiction that $x \equiv (i \cdot a) \mod p$, $x \equiv (j \cdot a) \mod q$ and $x \equiv (m \cdot a) \mod r$. Noting that $x \equiv 0 \mod a$ and $pq \mid r$, this implies $x/a \equiv i \mod p$, $x/a \equiv j \mod q$ and $x/a \equiv m \mod (p \cdot q)$. Therefore $i \equiv m \mod p$, and from $1 \le i, m \le k \le p$ we deduce $i = m$. Likewise, $j = m$, implying $i = j$.

Hence each iteration returns **true** with probability at least $1 - 1/k$. Since k independent iterations are run, the error probability is bounded by $1/k^k \le 1/(1000 \log n)$. ☐

C Upper bound for restricted oracle

C.1 Proof of Observation 6.6

Let us start assuming, for the sake of simplicity, that $\mathcal{A}$ is non-adaptive. Thus g is known at the start of the algorithm. Let $g = am + b$, where $a < m$. The algorithm can make queries to all positions of form cm for all $c \leqslant a$ at the start. This takes $\frac{g}{m}$ number of queries. Once all these queries are made, by the definition of the m-restricted oracle, for any $i \leqslant g$ the algorithm can obtain the value of $query(i)$ with at most one call to the m-restricted oracle. Since all the queries that the algorithm $\mathcal{A}$ makes are at most g, the new algorithm can simply simulate the algorithm $\mathcal{A}$ while making queries to the m-restricted oracle. Thus the total query complexity will be $q + \frac{g}{m}$.

Note that the algorithm does have to know g beforehand, as it can query position cm only when the algorithm $\mathcal{A}$ queries some i such that $i > cm$. Therefore the conversion works too when $\mathcal{A}$ is adaptive.

C.2 Proof of Lemma 6.9

For any set $A \subseteq \mathcal{P}_n$, we have $\nu_A(\mathrm{lcm}([n])) = \prod_{p \in A} p^{\lfloor \log_p(n) \rfloor} \le n^{|A|}$, so if we take $A_1, A_2, \dots,$ to be consecutive subsets of $\mathcal{P}_n \setminus A_0$ containing $\lfloor \log m / (2 \log n) \rfloor - 1$ primes each (except possibly the last), the second condition is satisfied for all

$i > 0$. Now the Prime Number Theorem implies that the number of sets in such a partition is $|\mathcal{P}_n|/(\log m/(2\log n)) = O(n/\log m)$ (which gives us the third condition), and also that $|A_0| = O(\sqrt{n}/\log n)$, implying $\nu_{A_0}(\mathrm{lcm}([n])) = 2^{O(\sqrt{n})} \leq \sqrt{m}$ for large enough c.

D Removing the need for upper bounds on r and s

In order to apply any of the algorithms presented so far, we need to be able to have at our disposal good upper bounds r' and s' on r and s. Given a candidate pair r' and s', Algorithms find_r_small_m(s,n) and find_r_large_m(s,n) can be used to build a deterministic procedure check(r',s') to decide if the bounds r' and s' are valid, i.e. $r \leq r'$ and $s \geq s'$. (Note that the probabilistic part of Algorithm 6, namely the call to find_period(s',r'), only comes into play when s' and r' are good bounds, and could in fact be skipped for this check). The query complexity of this check is $O(f(r')+g(s'))$, where

$$f(r') \triangleq \min\left(r'/m + \sqrt{r'},\; r'/\log m + \log r'/\log\log\log r' \right)$$

and

$$g(s') = s'/m.$$

We need to show that we can determine the precise values of s and r with $O(f(r)+g(s)+\log s)$ queries (Algorithm 10).

Algorithm 10 find_r_s() – find r and s

```
1: for$i = 1$ to $\infty$ do
2:     $r' = \lceil f^{-1}(2^i) \rceil$
3:     $s' = \lceil g^{-1}(2^i) \rceil$
4:     if check(s', r') then
5:         $r \leftarrow$ find_period(s', r')
6:         do a binary search on $[0,\infty)$ to find $s$
7:         return $(r,s)$
8:     end if
9: end for
```

Note that $f(r)$ and $g(s)$ are strictly increasing functions whose growth rate is bounded above by a polynomial. In particular their inverses satisfy $f(f^{-1}(x)+1) = O(x)$ and $g(g^{-1}(y)+1) = O(y)$. (We are viewing f and g as functions defined over the reals). Consider Algorithm 10. Clearly the call to $find(r',s')$ will succeed when i reaches the value $i_0 = \max(\lceil\log f(r)\rceil, \lceil\log g(s)\rceil)$. The number of queries made at this point is

$$\sum_{i=1}^{i_0} f(\lceil f^{-1}(2^i)\rceil) + g(\lceil g^{-1}(2^i)\rceil) = \sum_{i=1}^{i_0} O(2^i + 2^i),$$

which is $O(2^{i_0}) = O(\max(f(r), g(s))$. So we can determine r with $O(f(r)+g(s))$ queries. The next line finds s by binary search with $O(\log s)$ queries. This is possible because, once r is known, predicate check(r,s') can be computed by querying the two positions [7] s' and $s'+r$. (Note that the binary search must start from scratch and find tight upper bound on s, as the prior bound s' might be much larger than poly(s)). Therefore the precise values of r and s can be determined after $O(f(r)+g(s)+\log s)$ queries with high probability.

[7]Here Observation C.1 is implicitly used, coupled with the fact that position $r' \geq r$ has already been inspected at this point.